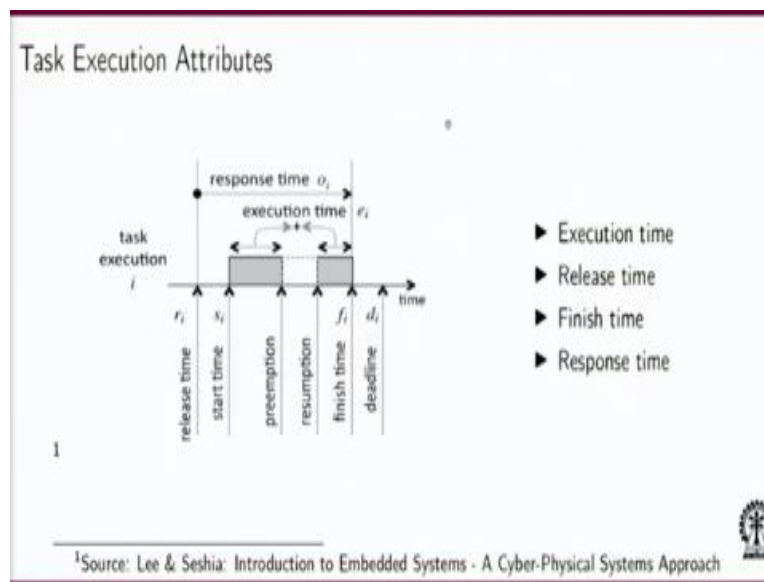


Foundation of Cyber Physical Systems
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 11
Real Time Task Scheduling for CPS (Continued)

Welcome back to this lecture series on Foundations of Cyber Physical Systems. So, if you remember in the previous lecture, we have been talking about real time task execution models and we will be continuing from there.

(Refer Slide Time: 00:39)



So, just to briefly introduce, what are the different task execution attributes that the real time task may have. So, this is the picture I have taken from this book by Edward Lee and Sanji Seshia on Introduction to Embedded Systems - A Cyber Physical Systems Approach. Actually, most of my coverage on real time systems is kind of reproduced from this literature. So, you can just refer to chapter 4 of this book on real time scheduling. You will see this picture.

So, typically when we talk about a periodic task. So, the characterization of a task will start with. So, this is a timeline and the time instant when the task is set forth for execution, that means it is

given the task has arrived and it has been released for execution is called the release time r_i . So, that means the system some dispatcher has provided the system with this task instance, okay. That does not mean that the task will start executing right now.

The task will start executing when the scheduler decides, the scheduler which is implemented as a may be set of interrupts through the hardware, or the scheduler which is implemented in the form of a software algorithm. I mean, that will decide when the task will really get access to the CPU and when it will really exit, right. So, that is what we call as the start time. When the task starts execution. And like we discussed earlier that the tasks execution may be preempted by some other event or some other task of higher priority.

So, these are points of preemption where a task can be preempted. And when the task resumes it will continue its further execution and in this way during its execution a task can be preempted multiple times and eventually when this task instance finishes, that is what we call as the finish time f_i . So, and this interval for starting from the release time to the actual finish time of the task is called the response time.

So, this is an important measurement of task execution, response time. Means when the task was made available and eventually when the task instance finished this interval is known as response time. And for a real time, task we always have an instance of deadline. So, let us say the deadline is given here as d_i and as you can see that the task finished before the deadline, right. So there is no lateness here.

So, when I am computing lateness, it is something which is basically finishing time minus the deadline. The lateness is negative that means the task executed within the deadline, okay. Now unless the deadline is explicitly specified, you can always assume that for a periodic task the period itself is the deadline. That means one is trying to say that as long as the next task instance from the same task has not arrived in the system, by this interval you must have executed the previous instance of the task which was dispatched to the system.

(Refer Slide Time: 03:56)

Scheduling types

- ▶ Hard real time (hard deadlines) | Soft real time (Soft deadlines)
- ▶ Non-Preemptive / Preemptive | Fixed/Dynamic Priority
- ▶ Priority – can be different than deadline. Can change / remain constant during task execution
- ▶ A **preemptive priority-based scheduler** supports arrivals of tasks and at all times is executing the enabled task with the highest priority.
- ▶ A **non-preemptive priority-based scheduler** uses priorities to determine which task to execute next after the current task execution completes.
 - never interrupts a task during execution to schedule another task.

Now just a small classification of different kinds of scheduling that happens in embedded and cyber physical systems. So, you can have a system which is a hard real time or a soft real time system. That means the deadline provided may be a hard deadline or a soft deadline. We have already talked about this preemptive and non-preemptive scheduling type. That means the scheduler may allow tasks to be preempted or not. And the other is tasks priorities.

If you remember we talked about static scheduling and dynamic scheduling, where task decisions were made statically offline before the system was online, or online scheduling where the task decisions were made during the systems execution. In a similar manner we can think of fixed and dynamic situation with respect to task priorities also. Now what is the priority of a task? See the moment we are talking about preemption and stuff like that we are saying that well one task can halt the, temporarily halt the execution of another task, and make itself to execute. That means the task which preempts another task will have a higher, I mean, some more importance, right. So, this idea of importance is being formally captured using the notion of priorities. So, every task can be thought of as having some priority level. Every task type can be thought of as having some priority level. So, let us say you have two sensors which are sampling data. Let us say lidar data and let us say camera data.

If your system considers the lidar data as more important to be processed, it will give, assign a higher priority to the lidar data, okay. So, when you are doing to going to do scheduling, eventually

it will be about priorities. Typically, in a dynamic scheduling system, if you are talking, going to implement preemption and other kind of things, it is about doing the priority assignment, that which task has higher priority that is going to be given the CPU.

Now the question is this kind of priorities, they can be like we have been discussing, they can be fixed or dynamic. That means for a specific task type, I can have a fixed priority. Or in another kind of scheduling I can allow tasks to have dynamic priority. That means, what is the priority of the task? See, what is the priority of the task is eventually going to decide which task gets the CPU, right. So, as long as typically a highest priority task has a released instance still not executed, that is what should get the CPU. Any scheduling algorithm will work like that. So, when we have any fixed priority scheduling algorithm, the idea is simple. You have many tasks with fixed priorities. Whenever a task of the highest priority is available it will execute, and so on so forth. Now in case the priorities are not fixed that means we are thinking that the priority will be dynamically calculated based on some task attribute.

Maybe how much time is left for the task to finish. Or maybe at what periodicity the task is going to come, assuming the periods may not be fixed, something like that. I mean, so, there may be some online situations. There may be some variations in periodicities of the tasks happening things like that. So, depending on such dynamic events if the task priority is decided, then we will say that well it is a dynamic priority scheduling algorithm.

Now, coming to the next point. Like we said earlier that priority can be different than deadline. If you remember earlier, we said that well if the deadline is not mentioned we can assume the deadline and period are same. But in general, for a given system, for full real time specification I will be measuring the period and also the deadline. Now one may start thinking that well if I have a task with an earlier deadline, that means I must execute earlier also. That is not really the case.

I mean it may be so, it may not be. So, I may have a situation where tasks are going to have different criticalities. So, I may have a task which is marked as critical and that may have a deadline far away. But since it is critical it may also have a higher priority. So, that is the point I am trying

to make, that priority in general may or may not be equivalent to deadline. Like we should not start suddenly thinking that everything which has the earliest deadline will always have the highest priority, because there may be some other attributes like we discussed.

So, based on these attributes. Like whether we are allowing preemption or not allowing preemption whether the priority is fixed or priority is computed dynamically at runtime, priority is assigned dynamically, we have these different classes of schedulers. We can have a preemptive priority-based scheduler. So, a preemptive priority-based scheduler will support arrival of tasks at all times, right. So, you have periodic tasks which are arriving.

And whenever a task, I mean, at any point of time you are looking at well, which task has the highest priority. Let us say you suddenly get a task inside your system which has priority more than whatever task is currently executing. You will be preempting that currently executing task and you will just execute this new task. So, let us say you have this CPU, and you are executing some task. So right now, if this task is executing, that means let us say you have T , T' which are waiting.

And let us say this is T'' , then priority of T'' must be greater than or equal to max of the priority of this T and priority of T' , right. That is why this is executing. Now suppose in this system you have a new task that has arrived, okay. And that means now, and let us say that task has a priority higher than this. Then this scheduling algorithm is very simple, right. It is just looking at its active set. And the set has been updated because a new task has arrived.

So, it will take a scheduling decision. And if it finds that current active set which has been updated is containing something having higher priority than whatever is executing, it may just preempt this task. And take it back to the incomplete task queue, and schedule whatever has arrived with the highest priority. Now the other option can be non-preemptive priority-based scheduler. So, the same thing like we discussed. But we do not preempt if anything is executing, okay.

So, based on priorities you are this going to decide that what should be the next task after the current one which is executing. That means you let the current one execute, finishes execution,

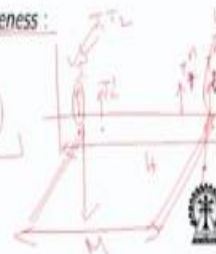
even if something with higher priority has arrived, okay. So, these two options come in. So, in this case what happens is you never interrupt a task during its execution to schedule another new task that may have arrived.

(Refer Slide Time: 11:02)

Optimality

- ▶ Scheduling goal : all task executions meet their deadlines, $f_i \leq d_i$
- ▶ Feasible schedule : any schedule which does the above
- ▶ Optimal w.r.t. feasibility : a scheduler that yields a feasible schedule for any task-set (if such a schedule really exists)
- ▶ Goodness of soft real-time schedulers : check maximum lateness :

$$L_{\max} = \max_{i \in T} (f_i - d_i)$$
- ▶ Another criteria : schedule makespan : $M = \max_{i \in T} f_i - \min_{i \in T} r_i$



So, if I want to formally characterize that what is my scheduling goal. If I am going to design a scheduler what is really the thing that this scheduler is going to do. The scheduler's job is simple. It is going to just check, that well, it is able to execute all these tasks before their deadlines. That means, the constraint that this scheduler must meet is for all tasks i belonging to this set of tasks. The finishing time is less than or equal to the deadline of each of the tasks, right.

So, observe that this may not be a unique thing, right. So, let us just empirically represent a schedule. Let us say you have some tasks T_1 and T_2 . They may be periodic. So, when I say that I have a scheduled sigma, what I mean is, I am able to assign to every instance of T_1 if even if it is repeating, I am able to assign a feasible start time. That well I say that this is when T_1 's instance one starts, this is when T_2 's instance one starts with this kind of a start time value. And this continues for all future instances.

If I can give a formula which is specifying these values and applying that formula, I can solve it for all future instances. Or I can just write a pattern that and follow that pattern of start times. That is my schedule. So, in general the way we look for schedules is that we try to schedule tasks up to

the hyper period. Now what is the hyper period? Let us say this task has period p_1 and this task has period p_2 . Hyper period H is nothing but the $lcm(p_1, p_2)$.

Now what would happen is inside this timeline of size H you have H/p_1 number of instances of the task T_1 and H/p_2 number of instances of the task T_2 , right. So, as long as you have a schedule which is able to figure out start times for all these instances, these number of instances of T_1 , and these number of instances of T_2 . This is what is going to repeat again in the next hyper period, right. So, if you try to draw up to some hyper period H , whatever instances of T_2 you have, you are able to schedule them and whatever instances of T_2 you have you are able to schedule them. This is what will repeat in the next hyper period and it is just going to continue, right. So, any schedule where for all these instances inside the hyper period you are able to satisfy this, you are going to satisfy this definitely in the next instance because that is what is just repeating, right. And there is a feasible schedule.

Now there is this terminology called optimal with respect to feasibility. So, what it means is a scheduler that yields a feasible schedule as long as it exists. So, I am saying optimal not with respect to execution time. Notice this. First, I have defined what is a feasible schedule. That means it satisfies the timing constraint. And then I am not saying, I am not saying optimal with respect to execution time. But I am saying optimal with respect to feasibility.

This means that as long as a feasible schedule exists, as long as a feasible schedule exists for a given task a given collection of task, my scheduler, if it is able to find that schedule which really exists, I would say that this is an optimal scheduler. But it is optimal with respect to what? It is optimal with respect to the feasibility condition that a schedule is really feasible and since I mean as long as it is feasible if I can give a guarantee that my scheduling algorithm is going to discover it and output it, I will say that this is my optimal algorithm. But it is not optimal with respect to time or throughput, it is optimal with respect to feasibility. What would be optimal with respect to time? It would simply be the scheduler which always gives me a schedule such that there exists no other scheduler which gives me another schedule which is going to have an overall time less than the one given by this scheduler. That would be something like optimal with respect to time.

Now there is something else if you remember we talked about not only real, I mean, hard deadline tasks, but we also talked about soft real time settings of tasks and schedules, right. So, in a hard deadline setting this needs to be satisfied, but in soft deadline deadlines may often get overrun, right. So, that means this quantity may be positive, right. So, this is what we call as lateness, right. So, when I am looking for soft real time systems it, I have to accept lateness, right. But we can create a metric which we call as this maximum lateness metric. That I figure out for all the tasks so what is the lateness that is incurred and I figure out what is the maximum among them. So, that would be like a metric to evaluate how good a soft real time schedule is. There is another criteria that we call a schedule makespan. So, I can also create a scheduling algorithms optimality check with respect to. Just like I we have defined it with respect to feasibility, it can be with respect to lateness, optimal with respect to lateness.

That means any other scheduling algorithm cannot give me a schedule which has a smaller lateness value, smaller max lateness value, okay. But if I am talking about it with respect to time, then the criteria really would be in a makespan. So, what is makespan? So, let us say you have a set of tasks, your schedule starts at several instances. Let us say your schedule is built on task T_1 and T_2 and let us say this is your hyper period inside which you have many instances of T_1 and many instances of T_2 . This is the earliest release time of this task T_1 and T_2 . Let us say T_1 's first instance got released here, T_2 's first instance got released here, right. So, among these two release times, this one is the earliest one, right. Now let us say the most late instance of T_1 got finished somewhere here, right. T_1^m let us say and the last instance of T_2 inside the hyper period got finished somewhere here, right.

So, the most late instance here is this, right. So, this is the largest finishing time, right. So, then I can typically say that inside the hyper period the overall task execution of all the tasks started at this point and they got finished somewhere at this point, right. And this interval is what I will like to call as the makespan, right **(18:40)**. So, then if I am trying to define this formally, I would say that among all the tasks finishing times the maximum value, which is this.

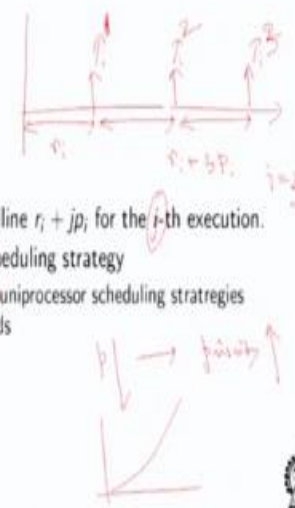
And from that you subtract that among all the instances release times, what is the minimum value. So that is the earliest position. So, subtract the earliest position from the late most position among

all the tasks, and this is the span inside which all your tasks have executed. And this is what I would let us say call it as a makespan. So, this was I will just repeat that T_1 is the last instance of T_1 . And T_1^1 was the first instance. And let us say all the T_2 instances are in between.

It could have been otherwise also like, this could have been T_2 and this could have been T_1 . Whatever is the last one to finish and whatever is the first one to start among all the tasks, right. That is why you have $\max_{i \in T} f_i - \min_{i \in T} r_i$. That is why the condition looks like this here. So, that is how we define makespan (19:48).

(Refer Slide Time: 19:50)

Rate Monotonic Scheduling (RMS)



- ▶ Set of tasks $T = \{\tau_1, \tau_2, \dots, \tau_n\}$
- ▶ Task τ_i with release time r_i , period p_i has deadline $r_i + jp_i$ for the i -th execution.
- ▶ RM by Liu and Leyland (1973): preemptive scheduling strategy
 - ▶ Optimal w.r.t. feasibility among fixed priority uniprocessor scheduling strategies
 - ▶ Gives higher priority to tasks with lower periods

Now let us get started with our first scheduling algorithm. So, we will start with one of the most popular yet most simple scheduling algorithm known as RMS or Rate Monotonic Scheduling. So, it was invented by Liu and Leyland, 1973. It is a preemptive scheduling strategy. That means that once the task set is updated the priorities will be checked and if there is a high priority task which recently arrive, it can preempt a current executing task, okay.

So, you are given a set of tasks and in this case, it is very simple. Priority is equal to the period. I mean if you have task with the smallest priority, it is going to have the highest period. I mean, task with the smallest period will have the highest priority, sorry. So, suppose you have a τ_i and this has got released at some time r_i . So, at time r_i you have got τ_i which has been released, right. And it has a period p_i . So, for some j th instance's execution, you have how many instances?

So, you have j number of instances and let us say these instances are separated by a period p , right. So, let us say $j = 2$ so this is the first instance, let. So let us call it the 0th instance here, right. And this is your second instance, right. Okay let us count for one only so τ_i . Let us count from 0. So, j is 2 that means you have 3 instances. This is this. And that say this is your $i = 1$ and that say this is your τ_i^2 . So, if you see what is the time like this is. So, these are the instances when the tasks are going to be released, and this is the initial release time. Or better this is known as the offset of the release, right. So, the eventual release time here will be $r_i + 3 * p_i$, right. If I am just counting from $i = 1$, so this is the third instance is 1, 2 and 3. So r_i plus 3 p_i is as simple as that. So, in that way given this initial offset I can always compute what is the exact time when these tasks some i th or j th instance is going to be released into the system, right.

Now similarly for this is what is happening for τ_1 . Similarly, you will have release instances for τ_2 . Similarly, you will have release instances for τ_3 . So and so forth. So this is the first release time. So, I will just repeat. Typically, this is what we call as the offset here. Now this algorithm, the way it is going to choose among task instances of the different tasks and give to the CPU, like we have discussed, is based on the priority and the priority is computed based on the rate of the task.

So, if a task has lower period value, it has a higher priority value. So, why do I call it Rate Monotonic? Lower period means higher rate, right. I mean, number of tasks executed is going to higher, the frequency of task is higher, right. So, with higher rate you have higher priority, okay. So, it is as simple as that. You have a set of tasks with varying periods the one with the highest rate or lowest period will always have the highest priority here.

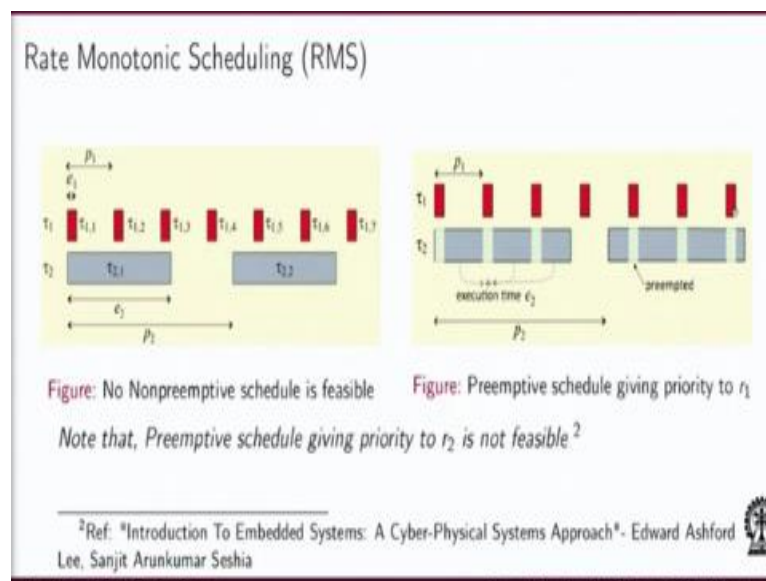
Now it can be shown that among all possible scheduling techniques which allow preemption in a uniprocessor scenario, and where the priority is fixed, this is optimal with respect to feasibility. So, let us understand again. So, see in this scheme the priority of the tasks do not change, right. Because the priority is simply defined as priority equal to, I mean, priority is just the I mean the inverse of period, I would say. I mean smaller the period the higher is the priority.

So, the higher the rate the higher is the priority. So, if you have a fixed task set with no change in periods, the priorities are never changing. So, as long as with a fixed set of tasks and their fixed attributes, the priorities are fixed. So, it is a fixed priority algorithm. And what we are saying is what if, for given any such task set, any such periodic task set, if I have any other fixed priority algorithm any algorithm which works with fixed priorities.

And they are able to find a feasible schedule, then this algorithm will also be able to find a feasible schedule. So, that is all about it. It is a very simple algorithm and it works as good as other algorithms, like this one says among the fixed priority algorithms. But why people love this algorithm is, it is very easy to implement. Because there is nothing much to implement. It is all about what are the tasks that are execute, I mean, inbound in the system, existing in the system. Which one has the lowest priority, the lowest period, sorry, just execute it.

The one with lowest period, if it is a there just execute it. It is as simple as that. So, it can be simply I mean, very, I mean implemented in a very lightweight manner and that is why still RM scheduling is extremely popular in real time systems.

(Refer Slide Time: 25:52)



So, we will just try to argue why this is optimal with respect to feasibility. So, let us have a look into this example. So, you have two tasks τ_1 and τ_2 and this, again this example has been taken from this book, okay, by Edward Lee and Sanji Seshia. So, you have τ_1 . As you can see that τ_1

has a smaller period it has a higher rate and τ_2 has a higher period a lower rate. e_1 is the execution time of τ_1 .

So, typically when I am now trying to characterize real time periodic tasks, they will be characterized by this kind of tuples, right. So, for some task T_1 , you have a period p_1 and the execution time and if the deadline is something different from period, you also write that, okay. So, for τ_1 you have this e_1 as the execution time, p_1 as the period and here you have e_2 as the execution time and p_2 as the period.

So, this is τ_2 second, first instance, and τ_2 second instance, okay. Now what we are trying to argue here is see, for this kind of a typical task set, it is not possible to have a non-preemptive schedule. I mean, there is no non primitive schedule which is feasible. So, let us understand why. I think it is quite evident from this picture. So, suppose you are not going to preempt the task, okay. Now suppose you are executing, I mean, initially these two both of them has come, right.

So, either you are executing to execute τ_1 or you are going to execute τ_2 , right. So, I mean let us say, I decide, I mean, if I am following Rate Monotonic. If I am trying to follow Rate Monotonic in a non-preemptive way, what am I really going to do? I am trying to execute tau 1 first. So, then once tau 1 is executed, from this point I am going to start executing τ_2 , right. So, essentially tau 2 will start from here and τ_2 will end somewhere here, right.

So, then see, since I am saying that I am going to having, I am going to have no preemption supported the moment τ_2 gets the CPU, τ_1 's other instances which will arrive during τ_2 's execution. For example, this instance and this instance, none of them will get access to the CPU. So, that means τ_1 will start missing deadline. So, you see that the non-preemptive schedule is simply not feasible.

Now let us look at the option of using preemption. With preemption this is pretty much solvable. With preemption if I start giving more priority to τ_1 , I have a feasible schedule. Feasible because all the task instances will execute before that deadline. So, see I, when I start, I will execute with

τ_1 and I am giving more priority to the more frequent task. Basically, I am following Rate Monotonic.

So, then I am starting τ_2 here, okay. This blue region I am executing. When τ_1 's next instance comes whenever τ_1 instance is available, I just preempt τ_2 , okay. So, in this way τ_2 will extend up to this much time. Whereas here τ_2 's execution time is supposed to be this much. But due to this it was actually accommodating all the instances of τ_1 . So, its execution time will actually extend by a specific amount and I think it is easy to calculate. We can just check that this is thrice of e_1 .

Because that is the time required to execute 3 instances of τ_1 . But as you see here τ_1 has always met your deadline. Because whenever it wanted the CPU, whenever it has been released, it has been immediately allocated at the CPU. Has τ_2 met its deadline? Yes. Because this is the deadline, when much before the deadline, this has executed, right. So that is fine. And similarly, this thing will just keep on continuing for the next hyper period, right.

Now you can just note that well, if I try to do preemptive schedule here by giving priority to τ_2 that is really not going to be feasible. So, this is not r_2 but rather this is τ_2 here. Because let us say I mean that is what we just discussed, right. If I am just going to give priority τ_2 , it is almost like this non preemptive scenario, we are talking about. That let us say, I decide that τ_2 will execute, okay. That means τ_2 has a higher priority.

So, whenever τ_1 is coming, τ_1 is not going to preempt τ_2 , because τ_2 has higher priority, right. So, all these instances will be missing their deadlines and well this may not but these two will definitely miss their deadlines, right. And then the schedule is not going to be feasible. So, we can actually go on to argue that while I mean I am in RMS optimal with respect to feasibility, but we will stop here for the interest of time. And we will continue in the next lecture from this point. Thank you.