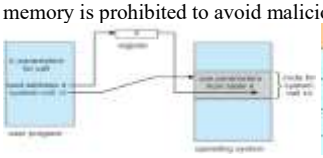
Real time system (Time constraints)
<Multi-task> <Schedulability> <Timely response is vital>
Clock <Correctness $|C(t) - Cs(t)| < \varepsilon$, Cs(t) is the standard clock, ε is granularity >
<Bounded drift, rate of change of clock away from perfect clock, $\left|\frac{dC(t)}{dt} - 1\right| < \rho\ (10^{-5})$>
<Monotonicity, $\forall t_2 > t_1; C(t_2) > C(t_1)$ Clock either increase or remain constant >
RTS <Soft RTS: performance is degraded but not destroyed by failure to meet response time constraints> <Firm RTS: missing few deadlines will not lead to total failure but missing more will>
<Hard RTS: Fail to meet a single deadline will lead to complete fail>

OS and Kernel (OS: System supervision, resource allocation, security, communication)
Normal workflow < User space ←→ System call ←→ Kernel space → hardware >
Kernel <Process & memory management, file system, scheduling, interrupt handling >
Kernel mode <All instruction allowed,including OS routings.Unlimited hardware access>
User mode <Only limited instruction is allowed. Direct access to hardware and the memory is prohibited to avoid malicious users>



	Operating System	Real-Time Operating System
Design philosophy	Time-sharing	Event-driven
Design requirement	High throughput	Schedulability
Performance metric	Fast average response	Limited worst-case response
Overhead	Fairness	Meet critical deadlines

Objectives of Non-real time OS <Fairness, efficiency, fast response time, throughtput >
Objectives of RTOS <Meet the deadlines, schedulability, worst-case response >
Implement RTOS based on Linux <Add new layer of RT kernel with control of interrupt and resources, make changes in Linux kernel to make suitable for RT applications>
Kernel module <extend the functionality of the kernel, save memory, no need to rebuild, reinstall, reboot the base kernel every time> <**Commands**: insmod, rmmod, lsmod, modinfo> <RT tasks running as kernel model need have **macro**: **module_init** (Invoked by insmod, use to allocate required system resources, declare and start tasks). Task specific code. **Function**: **cleanup_module** (rmmod, release all system resources allocation during module) > **Function**: init_module() | **Macro**: module_exit()
<**init / exit**: attribute used to optimize memory usage by marking functions as initialization or clean up only > <module_param(name, type, permissions): pass parameters >

```
1 static char *my_string __initdata = "dummy";
2 static int my_int __initdata = 4;
3
4 /* Init function with user defined name */
5 static int __init hello_4_init(void) {
6     printk(KERN_INFO "Hello %s world, number %i\n",
7     return 0;
8 }
```

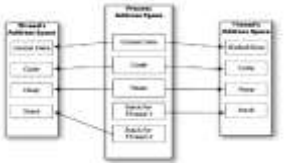
```
9  /* Exit function with user defined name */
10 static void __exit hello_4_exit(void) {
11     printk(KERN_INFO "Goodbye cruel world (%s)"
12 }
13
14 /* Macros to be used after defining init and e
15 module_init(hello_4_init);
16 module_exit(hello_4_exit);
```

Preemption & system calls
Why not make a raw system call rather than using API<API from library is easy to use, API provides security, API provides better portability, API can be more efficient>
Makefile <Compile multiple files, if some codes are changed, no need to compile all code again, but hard to track which parts need to be recompiled> <Only files depend on that files need to be recompiled >
Process <program in execution, abstraction of running program, logical unit of work scheduled by OS><Stack: store function arguments, local variables, return address of functions><Heap: Memory is dynamically by system calls, new, malloc, calloc><BSS segment: uninitialized data><Data segment: Initialized data, global and static variables><Text: Read-only region containing program instructions/code>

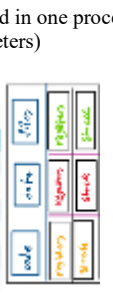
- if file1.c uses a function defined in file2.c
  - the function needs to be defined also in file2.h

```
$ gcc -c file1.c file2.c
$ ls
> file1.c file2.c file2.h file1.o file2.o
$ gcc file1.o file2.o
> file1.c file2.c file2.h file1.o file2.o a.out
```

Fork <Creates a child process identical to its parent>
In parent, fork() returns process ID of newly child process. In child process, returns 0
Threads (More efficient to implement tasks, subtasks can be implemented in one process)
Do not share stack memory (local variables, function call details, parameters)
Shared heap memory, global variables, code segments



	Interrupt	Polling
Speed	fast	slow
Efficiency	good	poor
CPU waste	low	high
Multitasking	yes	yes
Complexity	high	low
Debugging	difficult	easy



Pros of threads <Shared address space → communication among threads more efficient, context switching between threads in the same process is faster between processes, much quicker to create a thread that a process, thread programming is supported by POSIX>
<If **I/O bound** (in most cases), multiple threads will make the process more efficient>
Cons of threads <Need of synchronization → global variables are shared between threads: changes or modifications for shared variables can be pain, security: many library functions are not thread safe. Lack of robustness: If one thread crashes, the whole application crashes. >

- #include <pthread.h>
- Define a worker function:

```
void *function( void * );
```

- Initialize pthread attr t

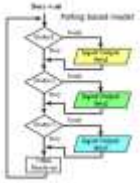
```
pthread_attr_t attr;
pthread_attr_init(&attr);
```

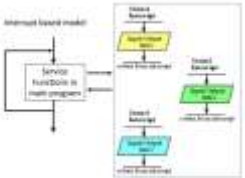
- Create a thread

```
pthread_t tid;
pthread_create(&thread, &attr, worker function, arg);
```

- Exit current thread

```
pthread_exit(status);
```





Race condition <Unpredictable and non-repeatable> <Happens when multiple process or threads access and manipulate shared resources variable concurrently> <Possible solution: Disable preemption/parallelization when scheduling process or use semaphores as atomic operation.> <In engineering, outcome should be predictable and repeatable>
Kill <Send a signal to a process > Signal <assign a signal to a function>
Polling: Constantly reading a memory location to receive updates of an event or input value. <Pros: Simple to write and debug, response time easy to determine> <Cons: Not sufficient to handle complex systems and waste of CPU time particularly when events polled occur>
Interrupt: Upon receiving an interrupt signal, the processor interrupts whatever it is doing and serves the request.

Linux priority and nice values
<Static priority ranged from 0 to 139, 0-99 are for real time task, 100-139 for users [-20,19] → nice values (Lower value higher priority) > <nice – 10 process: set process w/ priority w/ nice value 10> <nice -n -10 process: increase nice value by 10> <Need **privilege** to set higher prio of process, **unprivileged** user can only lower process prio>
<int setpriority(), set the priority to a specific value> <int nice(int inc): increase the process priority by inc>
Set scheduling policy in Linux <int sched_setscheduler(pid, policy, *param), policy is>
Need to be a privilege user to make this function call
Linux supported scheduling policies: **NON-RT**< SCHED_OTHER: the standard round-robin time-sharing policy; SCHED_BATCH: for batch style execution; SCHED_IDLE: for extra low prio background tasks;> **RT**<SCHED_FIFO; SCHED_RR; SCHED_DEADLINE: earliest deadline first>
RR in RTOS: applications mean weighted fair queueing (CPU time)
Creating a thread with a specified priority
struct sched_param has a field sched_priority which us used to set the process priority

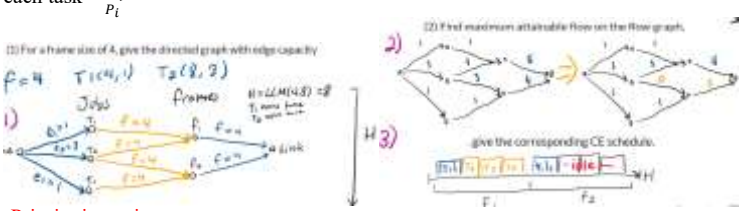
```
struct sched_param param;
param.sched_priority = 20;
```

Functions to set priority

```
pthread_attr_getschedparam (&attr, &param);
param.schedpriority = 10;
pthread_attr_setschedparam (&attr, &param);
// or
pthread_create(&thread, &attr, <worker function>, arg);
```

```
struct timespec{
    time_t tv_sec;
    long tv_nsec;
};
```

Sleep functions <**sleep**(): suspends thread> <**nanosleep**(*req, *rem):sleep but nanoseconds. If interrupt, rem = remaining time.> <**clock_nanosleep**(clockid, flag): greater control over time measurement. clockid can be CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID. Flags: 0 for interval, or TIMER_ABSTIME>
Real time tasks (concurrently, real time constraints)
Types of tasks<Periodic task: interarrival time is almost the same<Aperiodic task: irregularly task> <Sporadic task: aperiodic but with hard deadline>
Parameters <**n** number of tasks; $\boldsymbol{r_{i,j}}$ Release time or arrival time; $\boldsymbol{d_i}$ Absolute deadline; $\boldsymbol{D_i}$ Relative deadline; $\boldsymbol{E_i}$ Execution time; $\boldsymbol{R_i}$ Response time> <$\boldsymbol{d_i = r_i + D_i}$>
Task model for periodic tasks <Periodic, $D_i = p$, all tasks are independent, no task has any non-preemptible section and cost of preemption is negligible, CPU processing requirements are significant, memory and I/O requirements are negligible>
Representation of periodic tasks
T_i can be represented by 4 tuples (ϕ_i, P_i, e_i, D_i)
If using 3 tuples (P_i, e_i, D_i) → $(0, P_i, e_i, D_i)$
If using 2 tuples (P_i, e_i) → $(0, P_i, e_i, P_i)$
CPU utilization $u_i = e_i/p_i$
The system with n tasks over all system utilization is $U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{e_i}{p_i}$
Cyclic executive (CE)
Pros <Simple, no race condition, no deadlock> Cons<Brittle, any change requires a new table, F could be huge, release time of task must be fixed>
Assumptions <Parameters of jobs with hard deadline known> <Task scheduling is non-preemptive> <**Hyperperiod**: The least common multiple of periods> <**Max # of arriving jobs** in a hyperperiod is $N = \sum_{i=1}^{n} \frac{H}{p_i}$> <**Frames** each task must fit within a single frame, multiple tasks can execute in a signal frame; **frame size f** number of frames per H is $F = \frac{H}{f}$> <If a task must be completed every 5-time units but is delayed due to the structure of the frames, it can only start at the next frame boundary>
Frame size constraints <**C1:** A job must fit into a frame $f \geq \max_{1 \leq i \leq n} e_i$><**C2:** H must be evenly divided by f> <**C3:** f should be sufficiently small → $2f - \gcd(P_i, f) \leq D_i$>
Network flow problem <Number of jobs = $\sum \frac{H}{P_i}$> <Number of frames = $\frac{H}{f}$> <Number of each task = $\frac{H}{P_i}$>



Priority inversion
Mutual exclusion: Resource can be used by one task at one time
Serially reusable: User of a resource cannot be interrupted
Critical section: Code that interacts with serially reusable resource
<Avoid race condition by making sure no two processes enter cs at the same time, mutex lock but may lead to deadlock
Outermost cs: A cs that is not included in any other cs
Inversion: Even if the tasks are preemptable, resources are allocated on nonpreemptive basis → higher priority task can be blocked by lower priority task due to resource contention → why it happens (access shared resources by mutex, access non-preemptive subsystems, storage) → result in timing anomalies (some tasks can't meet deadline)

Rate monotonic (RM, shorter period tasks get higher priority, static priority)
Assumptions <tasks are running on uniprocessor system, tasks are preemptive, no OS overhead for preemption> <If two or more tasks have the same period, the scheduler selects one of these jobs random>
<Same lower priority tasks miss the deadlines every time and there is no effect on higher priority tasks and better predictability than EDF>
Schedulability test 1. $D_i \geq p_i$ for all processes, periods of all the tasks are integer multiple of each other. A **necessary and sufficient** condition for such tasks scheduled on a uniprocessor is $U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$

2. If the tasks have arbitrary periods, **a sufficient but not necessary** condition is $U \leq n(2^{\frac{1}{n}} - 1)$, as n goes to infinity, U goes to 0.692
3. A **sufficient and necessary** condition

$$\omega_i(t) = \sum_{k=1}^{i} \left\lceil \frac{t}{p_k} \right\rceil e_k \leq t \qquad 0 \leq t \leq p_i$$

$$t = k_j p_j, \quad (j = 1, \ldots, i) \qquad k_j = 1, \ldots, \left\lfloor \frac{p_i}{p_j} \right\rfloor$$

$$w = \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_1} \right\rceil e_2 + e_3$$

Start solving question from i = 1and find k.

Deadline monotonic (DM) → static priority
< Shorter relative deadline, higher priority > <**If every task has the period equal to relative deadline, then it's RM**> <For arbitrary deadlines, DM performs better than RM> <RM always fails if DM fails>
Earliest deadline first (EDF) → Dynamic priority scheduling (Priorities not fixed)
<The processor always executes the task whose absolute deadline d_i is earliest> <If 2 tasks have same d_i, EDF randomly selects one for execution next > <**Optimal** for uniprocessor scheduling> <**Cons**: EDF is hard to predict which tasks will miss their deadlines, high overhead, high complexity> <**Pros**: EDF is more flexible and has better utilization than RM> <A late task that has already miss its deadline has higher priority than a task whose deadline is still in the future>
Schedulability test 1. **Sufficient and necessary**: for tasks whose relative deadline is equal to or greater than its period. $\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$ if $D_i \geq p_i$

2. For $D_i < p_i$, a **sufficient** condition for such cases is $\sum_{i=1}^{n} \frac{e_i}{\min(D_i, p_i)} \leq 1$

MC. Given the task set: T1(4, 1); T2(5, 1); T3(10, 2), and you are asked to find a CE schedule using flow graph as shown in Figure 1. Assume that the frame size is 2. Which of the following is true? (There are 11 job nodes and 10 frame nodes in the flow graph.) **MC.** The __ scheduling algorithm schedules periodic tasks using a static priority policy with preemption (RM) **MC.** What does the term priority inversion refer to? (A situation where a high priority process must wait for a low priority process.) **MC.** How can a priority inversion be corrected? (Temporarily raising the priority of a low priority process.) **MC.** If a set of processes cannot be successfully scheduled by RM scheduling algorithm, then: (None of the above) **MC.** Consider the task set T = {(8, 4), (10, 2), (12, 3)}, which of the following is true: (T is not RM schedulable but EDF schedulable) **MC.** Which of the following protocol can handle the deadlock and priority inversion? (PCP) **MC.** A Task set consists of n pre-emptive and periodic tasks. If the task is NOT RM schedulable, which of the following is correct? (The CPU utilization is over 1) **MC.** What is the value of output value of g if the following code is given: (There is a race condition, so the value of g cannot be determined) **MC.** Which of the following is a drawback of thread programming? (Without synchronization, race condition on shared variables can be disastrous) **MC.** Which of the following output is not possible? Odd and Even code (1 0 2 4 5 3 6 7 9 8) **MC.** In CE scheduling, which of the following are correct? (Frame size of CE algorithm cannot be too small since we want an instance of a task completed within a single frame) **MC.** What is the output of the program? Note that SIGCHILD is sent to the parent and child process when it exits, is interrupted, or resumes after being interrupted. Handler and starts with val=9 (21). **MC.** Real-time systems must have __ (preemtive kernels) **MC.** Given 3 periodic tasks T1, T2, T3. They have the same execution time, but different periods. The periods of the tasks are 4, 8, and 16. What is the maximum execution time so that the tasks are RM schedulable (16/7) **MC.** Which of the following is NOT a benefit of using kernel module instead of installing all anticipated functionalities into a base kernel? (Allow preemption) **MC.** What is the output of the program? Swap a and b. (a=13, b=23) **MC.** What is the last line of the output program? foo, int a, static int sa. (a=15, sa=35) **MC.** When executing a C program, CPU runs in __ mode unless it is making a system call. (user) **MC.** Can child process access static variable created by a parent process before fork()? (Yes, but the modification can be seen only in child process, and the value in parent process will not be changed) **MC.** Multi-processor systems have advantage of __ over multi-thread systems. (reliability) **MC.** Which of the following is NOT shared by threads in the same process? (stack) **MC.** Which of the following is NOT true? (signal() function is used to send a signal to a process) (Actual truth is a field is updated in the signal table when the signal is sent) **MC.** For PCP, which of the following is true? (The system priority ceiling may change only when a resource is allocated or released) **MC.** Which of the following about the protocols to prevent priority inversion is correct? (Even if NPCS protocol is used, high priority process may still have to wait for a low priority process to release the resources) **MC.** A Task set consists of 5 pre-emptive and periodic tasks. If the task set is NOT RM schedulable which of the following can be inferred? (The CPU utilization is over 0.743) **MC.** Consider the task set T = {(P = 2, C = 1), (P = 4, C = 1), (P = 5, C = 1)}, which of the following is true? (T is both RM and EDF schedulable) **MC.** Which of the following statements about RM is correct? (Sufficient RM schedulability condition is based on processor utilization. It can be used at run-time to predict the schedulability but may lead to poor processor utilization) **MC.** Which of the following tasks are DM schedulable? (T1:(5, 1) T2:(8, 2) T3:(12, 4) and T4:(20, 2))

Solution 1: NPCS → Non-preemptive critical section protocol
Schedule all critical section non-preemptivelty. While a task holds a resource, it executes at a priority higher than the priorities of all tasks.
Pros: Does not need prior knowledge about resource requirements of tasks, simple to implement, can be used in both static and dynamic priority schedulers, good protocol when most critical sections are short and most tasks conflict with one another.
Cons: A task can be blocked by a lower priority task for long time without a resource conflict
Solution 2: PIP → Priority inheritance protocol → preemptible
Increase the priorities only upon resource contention. If lower priority task $T_L$blocks higher priority task T_H, priority(T_L) ← priority(T_H)
Transitive: T1 block by T2: pri(T2) ← pri(T1). T2 block by T3: pri(T3) ← pri(T1)
Pros: PIP avoid the cons of NPCS, PIP has most of the pros of NPCS
Cons: Does not avoid deadlock
Solution 3: PCP → priority ceil protocol
Extend PIP to prevent deadlocks and further reduce the blocking time
Assumptions: Assign priorities of all jobs are fixed, resource requirements of all the tasks will request a resource R is known.
Priority ceiling of a resource R(fixed): ceil(R) = highest priority among all tasks request R
Priority ceiling of a system (Dynamically changed): At any give time, a set of R are used, the highest priority ceiling among this set of resources.
Rules: If R is held by another task, the request fails and requesting task is blocked
If R is free: **1.** If the requesting task's priority is higher than the current priority ceiling of a system, R is allocated to it.
2. If the priority of the requesting task is not higher than the priority ceiling of the system, the request is denied, and task is blocked.
Exception: If the requesting task is holding a resource whose priority ceiling = the priority ceiling of the system, in which case the resource is allocated to the requesting task.