Lab 6: Processes scheduling

Innopolis University
Course of Operating Systems

Recap

- Name some of the different ways to implement IPC.

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.
- What is the difference between a process and a thread.

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.
- What is the difference between a process and a thread.
  - a process is defined as a task that is being completed by the computer, whereas a thread is a lightweight process that can be managed independently by a schedule

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.
- What is the difference between a process and a thread.
  - a process is defined as a task that is being completed by the computer, whereas a thread is a lightweight process that can be managed independently by a schedule
- How do we create a thread

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.
- What is the difference between a process and a thread.
  - a process is defined as a task that is being completed by the computer, whereas a thread is a lightweight process that can be managed independently by a schedule
- How do we create a thread
  - pthread_create()

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.
- What is the difference between a process and a thread.
  - a process is defined as a task that is being completed by the computer, whereas a thread is a lightweight process that can be managed independently by a schedule
- How do we create a thread
  - pthread_create()
- What is a race condition

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.
- What is the difference between a process and a thread.
  - a process is defined as a task that is being completed by the computer, whereas a thread is a lightweight process that can be managed independently by a schedule
- How do we create a thread
  - pthread_create()
- What is a race condition
  - when two or more threads access shared data concurrently, leading to unpredictable and often erroneous behavior

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.
- What is the difference between a process and a thread.
  - a process is defined as a task that is being completed by the computer, whereas a thread is a lightweight process that can be managed independently by a schedule
- How do we create a thread
  - pthread_create()
- What is a race condition
  - when two or more threads access shared data concurrently, leading to unpredictable and often erroneous behavior
- How can we avoid race conditions

Recap

- Name some of the different ways to implement IPC.
  - Pipes, Named Pipes, Message Queues, Shared Memory.
- What is the difference between a process and a thread.
  - a process is defined as a task that is being completed by the computer, whereas a thread is a lightweight process that can be managed independently by a schedule
- How do we create a thread
  - pthread_create()
- What is a race condition
  - when two or more threads access shared data concurrently, leading to unpredictable and often erroneous behavior
- How can we avoid race conditions
  - With Thread Synchronization.

Useful terms(1/2)

- **Burst time(BT)**, Burst time is the total time taken by the process for its execution on the CPU
- **Arrival time(AT)**, is the time when a process enters into the ready state and is ready for its execution.
- **Exit time(ET)**, is the time when a process completes its execution and exits from the system.
- **Response time(RT)**, is the time spent when the process is in the ready state and gets the CPU for the first time.
  - RT = Time at which the process gets the CPU for the first time - AT

Useful terms(2/2)

- **Waiting time(WT)**, is the total time spent by the process in the ready state waiting for CPU

$$WT = TAT - BT$$

- **Turnaround time(TAT)**, is the total amount of time spent by the process from coming in the ready state for the first time to its completion

$$TAT = BT + WT$$

$$TAT = ET - AT$$

- **Throughput**, is a way to find the efficiency of a CPU. It can be defined as the number of processes executed by the CPU in a given amount of time

Preemptive Scheduling

- Preemptive scheduling is used when a process switches from the running state to the ready state or from the waiting state to the ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.
- Algorithms based on preemptive scheduling are Round Robin (RR), Shortest Remaining Time First (SRTF), Shortest Job First (preemptive version), Priority (preemptive version), etc.

Non-Preemptive Scheduling

- Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non-preemptive scheduling does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then it can allocate the CPU to another process.
- Algorithms based on non-preemptive scheduling are: First Come First Served, Shortest Job First (nonpreemptive version), and Priority (nonpreemptive version), etc.

First come, first served (1/2)

- First come – First served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes according to the order they arrive in the ready queue. In this algorithm, the process that comes first will be executed first and next process starts only after the previous gets fully executed.
- It is non-preemptive.
- Average Waiting Time is not optimal
- Cannot utilize resources in parallel: Results in Convoy effect (Consider a situation when many IO bound processes are there and one CPU bound process. The IO bound processes have to wait for CPU bound process when CPU bound process acquires CPU. The IO bound process could have better taken CPU for some time, then used IO devices).

First come, first served (2/2)

Algorithm:

1. Input the processes along with their burst time (bt).
2. Find waiting time (wt) for all processes.
3. As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. $\text{wt}[0] = 0$.
4. Find waiting time for all other processes i.e. for process i: $wt[i] = bt[i-1] + wt[i-1]$.
5. Find turnaround time = waiting_time + burst_time for all processes.
6. Find average waiting time = total_waiting_time / no_of_processes.
7. Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

Shortest job next(1/3)

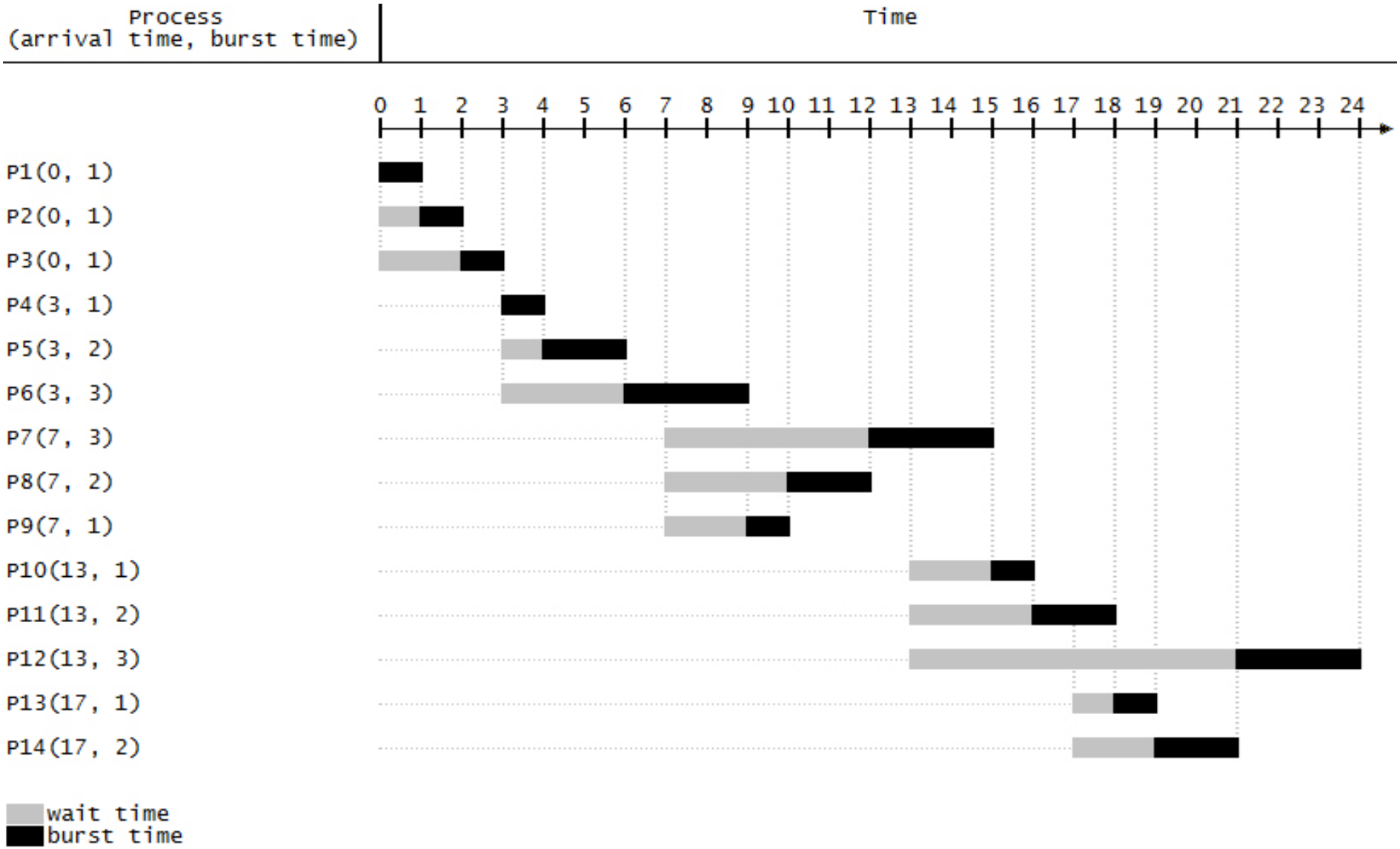
- The shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN, also known as Shortest Job Next (SJN).
- it can be preemptive or non-preemptive.
- minimum average waiting time among all scheduling algorithms.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- It is practically infeasible as Operating System may not know burst times and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.
- SJF can be used in specialized environments where accurate estimates of running time are available.

Shortest job next(2/3)

Algorithm:

1. Sort all the processes according to the arrival time.
2. Then select that process that has minimum arrival time and minimum Burst time.
3. After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

Shortest job next(3/3)



Round robin(1/3)

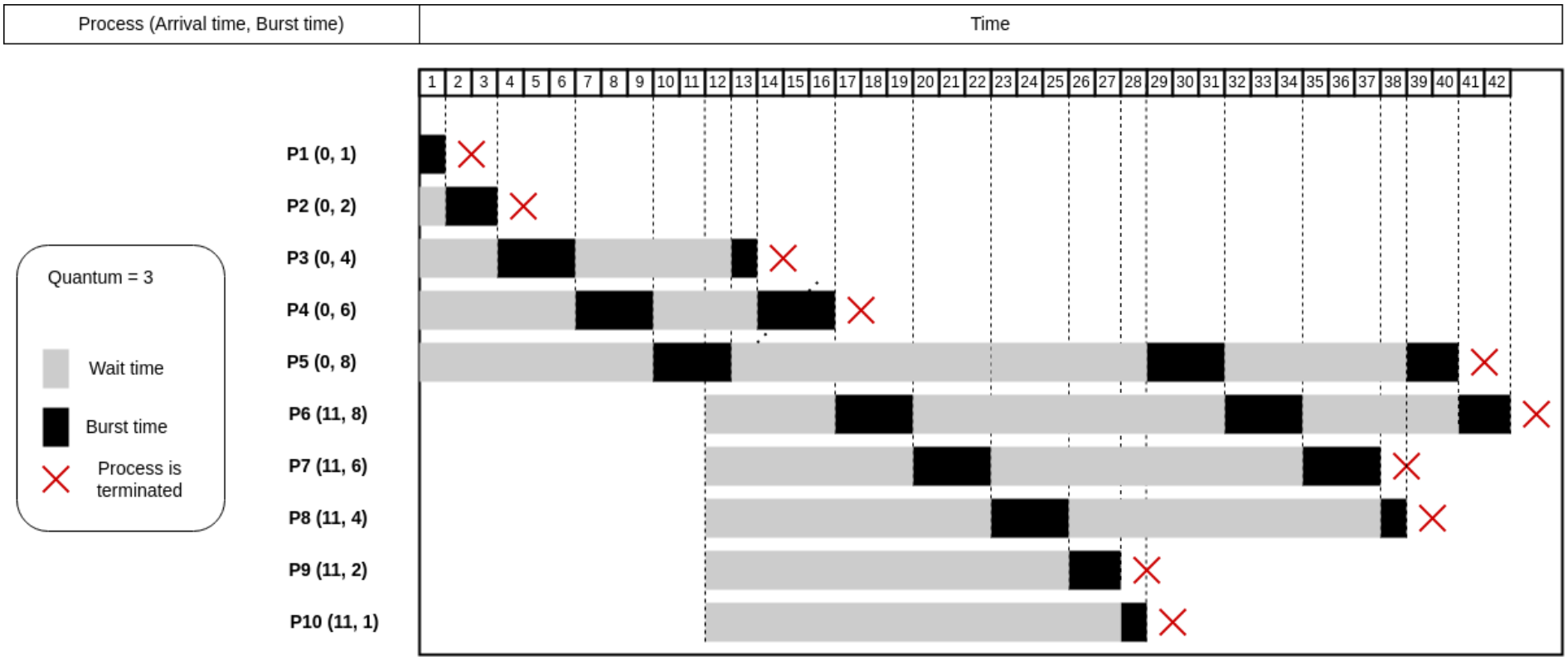
- Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of the First come First Serve CPU Scheduling algorithm.
- Round Robin CPU Algorithm generally focuses on Time Sharing technique.
- The period of time for which a process or job is allowed to run in a pre-emptive method is called time quantum.
- Each process or job present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will end else the process will go back to the waiting table and wait for its next turn to complete the execution.

Round robin(2/3)

Algorithm:

1. Create an array rem_bt[] to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)
2. Create another array wt[] to store waiting times of processes. Initialize this array as 0.
3. Initialize time : t = 0
4. Keep traversing all the processes while they are not done. Do following for i'th process if it is not done yet.
   - If rem_bt[i] > quantum
     - t = t + quantum
     - rem_bt[i] -= quantum;
   - Else // Last cycle for this process
     - t = t + rem_bt[i];
     - wt[i] = t – bt[i]
     - rem_bt[i] = 0; // This process is over

Round robin(3/3)



Signals (1/7)

- A **signal** is a software interrupt delivered to a process. It is an event that is generated to notify a process or thread that some important situation has arrived.
- When a process or thread has received a signal, the process or thread will stop what it is doing and take some action (not always). Signals can be useful for inter-process communication.
- **Standard signals:** The signals are defined in the header file **signal.h** as macro constants.
- Signal name starts with a "SIG" and is followed by a short description of the signal (every signal also has a unique numeric value). Your program should always use the name of the signals, not the signal numbers. The reason is that the signal number can differ according to the system or the implementation of the kernel but the meaning of names will be standard.
- The macro "NSIG" is one greater than the total number of signals defined. IF "NSIG" is 65 then there are 64 signals.

Signals (2/7)

```
firasj@firasj-vm:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Standard signals in Ubuntu 22.04.03

Source: https://www.gnu.org/software/libc/manual/html_node/Standard-Signals.html

Signals (3/7)

Some common signals:

- *int* SIGINT (Termination signal)
  - A "program interrupt" signal is sent when the user types the INTR character (normally Ctrl+c) to all foreground processes that are controlled by the terminal. The process that receives this signal is normally forced to terminate, but the process may arrange to ignore the signal or call a signal-catching function (signal handler).
- *int* SIGKILL (Termination signal)
  - This signal is used to cause immediate program termination. It cannot be handled or ignored and is therefore always fatal. It is also not possible to block this signal.
  - In fact if SIGKILL fails to terminate a process, that by itself constitutes an operating system bug that you should report.

Signals (4/7)

- *int* SIGABRT (Program error signal)
  - This signal indicates an error detected by the program itself and reported by calling "**void abort(void)**" system call.
- *int* SIGALRM (Alarm signal)
  - This signal typically indicates the expiration of a timer that measures real or clock time. It is usually used by the "**unsigned int alarm(unsigned int seconds)**" system call.
- *int* SIGQUIT (Termination signal)
  - This signal is similar to **SIGINT**, except that it is controlled by a different key—the QUIT character (usually Ctrl+\). It produces a core dump file that records the state of the process at the time of termination. The core dump file is named *core* and is written in whichever directory is current in the process at the time.
- *int* SIGTERM (Termination signal)
  - This signal is a generic signal used to cause program termination. Unlike **SIGKILL**, this signal can be blocked, handled, and ignored. It is the normal way to politely ask a program to terminate. The shell command "kill" generates **SIGTERM** by default.

Signals (5/7)

- *int* SIGSEGV (Program error signal)
  - This signal is generated when a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read. (Actually, the signals only occur when the program goes far enough outside to be detected by the system's memory protection mechanism.) The name is an abbreviation for "segmentation violation". Common ways of getting a **SIGSEGV** condition include dereferencing a null or uninitialized pointer, or when you use a pointer to step through an array, but fail to check for the end of the array.
- *int* SIGBUS (Program error signal)
  - This signal is generated when an invalid pointer is dereferenced. Like **SIGSEGV**, this signal is typically the result of dereferencing an uninitialized pointer. The difference between the two is that **SIGSEGV** indicates an invalid access to valid memory, while **SIGBUS** indicates an access to an invalid address. It varies among systems whether dereferencing a null pointer generates **SIGSEGV** or **SIGBUS**.

Signals (6/7)

- *int* SIGSTOP (Job control signal)
  - This signal stops the process. It cannot be handled, ignored, or blocked. Notice that SIGTSTP signal is an interactive stop signal and can be handled and ignored.
- *int* SIGCONT (Job control signal)
  - You can send a SIGCONT signal to a process to make it continue. This signal is special—it always makes the process continue if it is stopped before the signal is delivered. The default behavior is to do nothing else. You cannot block this signal. You can set a handler, but SIGCONT always makes the process continue regardless.
- *int* SIGUSR1 (Miscellaneous signal)
- *int* SIGUSR2 (Miscellaneous signal)

  The SIGUSR1 and SIGUSR2 signals are set aside for you to use any way you want. They're useful for simple interprocess communication if you write a signal handler for them in the program that receives the signal. The default action is to terminate the process. You can send signals via `int raise(int signum)` or `int kill(pid_t pid, int signum)`.

Signals (7/7)

Example:

- Write a C program that prints "Hello world!" an infinite number of times.
- Run the program, and kill it by pressing (Ctrl+C). Which signal did the process receive? Can we handle it? Can the process ignore the signal?
- Run the program, and kill it by pressing (Ctrl+\). Which signal did the process receive? Can we handle it? Can the process ignore the signal? In Ubuntu, set core dump size via `ulimit -c unlimited`. The core dump file is in */var/lib/apport/coredump/*. Check the log file */var/log/apport.log* for core dump file name.
- Run the program in the background (&) and redirect *stdout* to a file *output.txt*, kill the process by sending the signal "SIGTERM" via "**kill -s SIGTERM** *pid*" command. Can we handle the signal? Can the process ignore the signal? Send the signal "SIGKILL" if the program does not terminate politely.

Signal handling (1/4)

Each signal has a default action which could be one of the following:

- Term: The process will terminate.
- Core: The process will terminate and produce a core dump file.
- Ign: The process will ignore the signal.
- Stop: The process will stop.
- Cont: The process will continue from being stopped.

Default action may be changed using signal handler functions. Some signal's default action cannot be changed (e.g. SIGKILL, SIGSTOP).

- The simplest way to change the action for a signal is to use `signal` function. You can specify a built-in action (such as to ignore the signal), or you can establish a handler. The GNU C Library also implements the more versatile `sigaction` facility.

Signal handling (2/4)

The **signal** function provides a simple interface for establishing an action for a particular signal. The function and associated macros are declared in the header file **signal. h**. The signal function is declared as:

```
sighandler_t signal (int signum, sighandler_t action)
```

- The first argument, signum, identifies the signal whose behavior you want to control and should be a signal number.
- The second argument, action, specifies the action to use for the signal signum. This can be one of the following:
  - **SIG_DFL**: the default action for the particular signal.
  - **SIG_IGN**: the signal should be ignored.
  - *handler*:
    - signal handlers take one integer argument specifying the signal number, and have return type void. So, you should define handler functions like this: `void handler (int signum) { ... }`

Note: The **signal** function returns the action that was previously in effect for the specified **signum** or SIG_ERR in case of errors.

Signal handling (3/4)

Example:

- Run the "Hello world" program an infinite number of times and terminate it using **SIGINT**. Write a handler that prints the message "Interrupted!". Kill it using **SIGQUIT** or **SIGKILL**.
- Modify the program to let it print the message for the first time but apply the default action of **SIGINT** for the next time.
- Use **alarm** function to set a timer for the child process and let the child process print the message only for 3 seconds then terminate it. Which signal did the child process receive after 3 seconds? Write a handler for this signal which prints the pid of the child process.

Signal handling (4/4)

- Write a program that accepts a positive integer from stdin. The program raises the signal **SIGUSR1** if the input starts with '-' (could be a negative number) and signal **SIGUSR2** if the input is not a number (use `isdigit` function from `ctype.h`). Write signal handlers that print some useful information for the user.

Exercise 1 (1/2)

- Write two programs **agent.c** (Agent) and **controller.c** (Controller) and create a file **text.txt** which contains some message.
- On startup, the agent program creates the file `/var/run/agent.pid` and writes its `PID` into the file. It then reads the contents of a file called **text.txt**, prints it on the console and finally sleeps indefinitely.
- The agent would be sensitive to two signal types: SIGUSR1 and SIGUSR2. When the agent receives a SIGUSR1, it reads the contents of the text from **text.txt** and prints it on the console; when it receives a SIGUSR2, it prints "Process terminating..." and then exits.

Exercise 1 (2/2)

- On startup, the controller checks for a running agent by fetching the agent's PID from the file `/var/run/agent.pid`. If it cannot find an agent, it prints "Error: No agent found." and then exits; otherwise, it prints "Agent found." and continues.
- Then in an infinite loop, the controller prints "Choose a command {"read", "exit", "stop", "continue"} to send to the agent" and then waits for user input. The user enters one of the commands.
- When the user enters his/her choice, the controller sends the corresponding signal {"read":"SIGUSR1", "exit":"SIGUSR2", "stop":"SIGSTOP", "continue":"SIGCONT"} to the agent. If the user chooses "exit", the controller should also terminate after sending the signal. If the user presses Ctrl+C, the controller should send a SIGTERM to the agent and then exit.
- Submit **agent.c** and **controller.c**.

Exercise 2 (1/15)

worker.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <unistd.h>
#include <signal.h>
#define TRI_BASE 1000000

// current process pid
pid_t pid;
// current process idx (starts from 0)
int process_idx;
// number of triangular numbers found
long tris;
// checks if n is triangular
bool is_triangular(int n){
    for (int i = 1; i <= n; i++){
        if (i * (i + 1) == 2 * n){
            return true;
        }
    }
    return false;
}
```

```c
// generates a big rand number n
long big_n() {
    time_t t; long n = 0;
    srand((unsigned) time(&t));
    while(n < TRI_BASE) n += rand();
    return n % TRI_BASE;
}

// Given: signal handler
void signal_handler(int signum);
```

Exercise 2 (2/15)

scheduler.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

typedef struct {
    int idx;
    int at, bt, rt, wt, ct, tat;
    int burst; // remaining burst
} ProcessData;

// idx of running process
// -1 means no running processes
int running_process = -1;
// the total time of the timer
unsigned total_time;
// data of the worker processes
ProcessData data[PS_MAX];
// array of all worker processes
pid_t ps[PS_MAX];
// size of data array
unsigned data_size;

//TODO: read data from file and store
    it in data array
void read_file(FILE* file);

//TODO: send signal SIGCONT to worker
    process
void resume(pid_t process);
//TODO: send signal SIGTSTP to worker
    process
void suspend(pid_t process);
//TODO: send signal SIGTERM to worker
    process
void terminate(pid_t process);

//TODO: create a process using fork
void create_process(int new_process);

// TODO: find next process for running
ProcessData find_next_process();

// Given: reports the metrics and
    simulation results
void report();

// Given: scheduler waits for all
    processes
void check_burst();

// TODO: schedule processes
void schedule_handler(int signum);
```

Exercise 2 (3/15)

- Assume that we have only one CPU with one core. Only one process can run simultaneously and no race conditions might occur. You **should** use the code templates in this gist. The code snippets in the previous slides contain only function signatures.
- Write a program **scheduler.c** (scheduler) to simulate the first come, first served (FCFS) algorithm using a timer. Write another program **worker.c** (worker process) which is responsible for finding and counting the **next** triangular numbers from the range (n, ∞) where n is a big number randomly generated by the function $long\ big\_n()$.
- In this simulation, each process has an index (idx), arrival time (at) and burst time (bt). The scheduler reads the data of all processes from a file $data.txt$ where it contains $(m + 1)$ rows and 3 columns **(idx, at, bt)**. Here, we can have at most 10 processes where the actual number of processes is m.

Exercise 2 (4/15)

- Then, the scheduler runs a timer (`struct itimerval`) and sends a signal *SIGALRM* every second. You need to use a timer and set it to decrement in real time. You need to write the function *schedule_handler* which schedules the processes according to the given algorithm, in this exercise we have FCFS algorithm.
- The worker process **(worker.c)** in the main function will read its **idx** as a command line argument. Then it registers the same handler **signal_handler** for all three signals (SIGTSTP, SIGCONT, SIGTERM). The worker process generates a number n using **big_n** function. In an infinite loop, the worker checks the candiadate for the next triangular number **next_n** using **is_triangular**, when it finds the triangular number **next_n**, it needs to increments the counter **tris** and then checks again for the next number **(next_n + 1)** till termination by the scheduler.

Exercise 2 (5/15)

- The function `create_process`, firstly, stops any running workers, then, forks a new worker process and runs an instance of worker program **worker.c** using **execvp** where the argument `new_process` is the idx of the new process and passed as a command line argument to the worker program.
- The function `find_next_process`, will return the **ProcessData** of the *next process to run* according to the algorithm. In FCFS, you need to search in the **data** array for the next process based on their arrival time. In case, more than one process have the same arrival time then we can take the process with smallest idx as the next process to run. In case there are no processes arrived yet, then you need to find the next process again after incrementing the `total_time` till finding one process.
- The shcheduler terminates when the burst time of all processes becomes zero and prints a report before termination.

Exercise 2 (6/15)

- The implementation should calculate the following metrics and add to the **ProcessData** of each process: (**Note:** Go to the last slides to see the formula of the required metrics)
  - Response time (RT)
  - Completion time(CT)
  - Turn around time(TAT)
  - Waiting time(WT)
  - Average Turnaround time
  - Average waiting time
- Based on your implementation, answer the following questions and add your answers to a file **ex2.txt**.
  - Change the macro **TRI_BASE** to a small number like 100 and run the scheduler again. Compare your results before and after the change?
  - Change the arrival time of all processes to make all of them zero, then run the scheduler again. In which order will processes be executed?

Exercise 2 (7/15)

- Submit **worker.c**, **scheduler.c**, **ex2.txt** with a script **ex2.sh** to run the program.
- **Note:** The user runs only the scheduler program passing **data.txt** as a command line argument, then the scheduler will run workers based on the given data. Do not forget to compile the **worker.c** before running **scheduler.c**.

Exercise 2 (8/15)

Example: The input is a **data.txt** file as follows:

```
idx at bt
0   5    2
1        7          6
2        20    3
3        3   8
4   2                    4
5   3                    1
6   10                    6
```

which can be visualized as a table:

idx	at	bt
0	5	2
1	7	6
2	20	3
3	3	8
4	2	4
5	3	1
6	10	6

The output is in the next slides.

(Note: Some screenshots are taken intentionally starting from the last line of the previous screenshot for continuity)

Exercise 2 (9/15)

```
Scheduler: Runtime: 1 seconds.
Process 4: has not arrived yet.
Scheduler: Starting Process 4 (Remaining Time: 4)
Process 4 (PID=<252152>): has been started
Process 4 (PID=<252152>): will find the next trianguar number from (954860, inf)
Scheduler: Runtime: 3 seconds.
Scheduler: Process 4 is running with 3 seconds left
Process 4 (PID=<252152>): I found this trianguar number 955653
Scheduler: Runtime: 4 seconds.
Scheduler: Process 4 is running with 2 seconds left
Scheduler: Runtime: 5 seconds.
Scheduler: Process 4 is running with 1 seconds left
Scheduler: Runtime: 6 seconds.
Scheduler: Process 4 is running with 0 seconds left
Scheduler: Terminating Process 4 (Remaining Time: 0)
Process 4 (PID=<252152>): count of triangulars found so far is 1
Process 4: terminating....
Scheduler: Starting Process 3 (Remaining Time: 8)
Process 3 (PID=<252153>): has been started
Process 3 (PID=<252153>): will find the next trianguar number from (353906, inf)
Process 3 (PID=<252153>): I found this trianguar number 354061
Process 3 (PID=<252153>): I found this trianguar number 354903
Scheduler: Runtime: 7 seconds.
Scheduler: Process 3 is running with 7 seconds left
Process 3 (PID=<252153>): I found this trianguar number 355746
Scheduler: Runtime: 8 seconds.
Scheduler: Process 3 is running with 6 seconds left
Process 3 (PID=<252153>): I found this trianguar number 356590
Process 3 (PID=<252153>): I found this trianguar number 357435
Scheduler: Runtime: 9 seconds.
Scheduler: Process 3 is running with 5 seconds left
Process 3 (PID=<252153>): I found this trianguar number 358281
Process 3 (PID=<252153>): I found this trianguar number 359128
```

Exercise 2 (10/15)

```
Process 3 (PID=<252153>): I found this trianguar number 359128
Scheduler: Runtime: 10 seconds.
Scheduler: Process 3 is running with 4 seconds left
Process 3 (PID=<252153>): I found this trianguar number 359976
Scheduler: Runtime: 11 seconds.
Scheduler: Process 3 is running with 3 seconds left
Process 3 (PID=<252153>): I found this trianguar number 360825
Process 3 (PID=<252153>): I found this trianguar number 361675
Scheduler: Runtime: 12 seconds.
Scheduler: Process 3 is running with 2 seconds left
Process 3 (PID=<252153>): I found this trianguar number 362526
Process 3 (PID=<252153>): I found this trianguar number 363378
Scheduler: Runtime: 13 seconds.
Scheduler: Process 3 is running with 1 seconds left
Process 3 (PID=<252153>): I found this trianguar number 364231
Scheduler: Runtime: 14 seconds.
Scheduler: Process 3 is running with 0 seconds left
Scheduler: Terminating Process 3 (Remaining Time: 0)
Process 3 (PID=<252153>): count of triangulars found so far is 13
Process 3: terminating....
Scheduler: Starting Process 5 (Remaining Time: 1)
Process 5 (PID=<252154>): has been started
Process 5 (PID=<252154>): will find the next trianguar number from (110156, inf)
Process 5 (PID=<252154>): I found this trianguar number 110215
Process 5 (PID=<252154>): I found this trianguar number 110685
Process 5 (PID=<252154>): I found this trianguar number 111156
Process 5 (PID=<252154>): I found this trianguar number 111628
Process 5 (PID=<252154>): I found this trianguar number 112101
Process 5 (PID=<252154>): I found this trianguar number 112575
Process 5 (PID=<252154>): I found this trianguar number 113050
Process 5 (PID=<252154>): I found this trianguar number 113526
Scheduler: Runtime: 15 seconds.
Scheduler: Process 5 is running with 0 seconds left
```

Exercise 2 (11/15)

```
Scheduler: Process 5 is running with 0 seconds left
Scheduler: Terminating Process 5 (Remaining Time: 0)
Process 5 (PID=<252154>): count of triangulars found so far is 8
Process 5: terminating....
Scheduler: Starting Process 0 (Remaining Time: 2)
Process 0 (PID=<252155>): has been started
Process 0 (PID=<252155>): will find the next trianguar number from (520288, inf)
Process 0 (PID=<252155>): I found this trianguar number 520710
Scheduler: Runtime: 16 seconds.
Scheduler: Process 0 is running with 1 seconds left
Process 0 (PID=<252155>): I found this trianguar number 521731
Scheduler: Runtime: 17 seconds.
Scheduler: Process 0 is running with 0 seconds left
Scheduler: Terminating Process 0 (Remaining Time: 0)
Process 0 (PID=<252155>): count of triangulars found so far is 2
Process 0: terminating....
Scheduler: Starting Process 1 (Remaining Time: 6)
Process 1 (PID=<252156>): has been started
Process 1 (PID=<252156>): will find the next trianguar number from (121458, inf)
Process 1 (PID=<252156>): I found this trianguar number 121771
Process 1 (PID=<252156>): I found this trianguar number 122265
Process 1 (PID=<252156>): I found this trianguar number 122760
Process 1 (PID=<252156>): I found this trianguar number 123256
Process 1 (PID=<252156>): I found this trianguar number 123753
Process 1 (PID=<252156>): I found this trianguar number 124251
Process 1 (PID=<252156>): I found this trianguar number 124750
Scheduler: Runtime: 18 seconds.
Scheduler: Process 1 is running with 5 seconds left
Process 1 (PID=<252156>): I found this trianguar number 125250
Process 1 (PID=<252156>): I found this trianguar number 125751
Process 1 (PID=<252156>): I found this trianguar number 126253
Process 1 (PID=<252156>): I found this trianguar number 126756
Process 1 (PID=<252156>): I found this trianguar number 127260
```

Exercise 2 (12/15)

```
Process 1 (PID=<252156>): I found this trianguar number 127260
Process 1 (PID=<252156>): I found this trianguar number 127765
Scheduler: Runtime: 19 seconds.
Scheduler: Process 1 is running with 4 seconds left
Process 1 (PID=<252156>): I found this trianguar number 128271
Process 1 (PID=<252156>): I found this trianguar number 128778
Process 1 (PID=<252156>): I found this trianguar number 129286
Process 1 (PID=<252156>): I found this trianguar number 129795
Process 1 (PID=<252156>): I found this trianguar number 130305
Process 1 (PID=<252156>): I found this trianguar number 130816
Scheduler: Runtime: 20 seconds.
Scheduler: Process 1 is running with 3 seconds left
Process 1 (PID=<252156>): I found this trianguar number 131328
Process 1 (PID=<252156>): I found this trianguar number 131841
Process 1 (PID=<252156>): I found this trianguar number 132355
Process 1 (PID=<252156>): I found this trianguar number 132870
Process 1 (PID=<252156>): I found this trianguar number 133386
Process 1 (PID=<252156>): I found this trianguar number 133903
Process 1 (PID=<252156>): I found this trianguar number 134421
Scheduler: Runtime: 21 seconds.
Scheduler: Process 1 is running with 2 seconds left
Process 1 (PID=<252156>): I found this trianguar number 134940
Process 1 (PID=<252156>): I found this trianguar number 135460
Process 1 (PID=<252156>): I found this trianguar number 135981
Process 1 (PID=<252156>): I found this trianguar number 136503
Process 1 (PID=<252156>): I found this trianguar number 137026
Process 1 (PID=<252156>): I found this trianguar number 137550
Scheduler: Runtime: 22 seconds.
Scheduler: Process 1 is running with 1 seconds left
Process 1 (PID=<252156>): I found this trianguar number 138075
Process 1 (PID=<252156>): I found this trianguar number 138601
Process 1 (PID=<252156>): I found this trianguar number 139128
Process 1 (PID=<252156>): I found this trianguar number 139656
```

Exercise 2 (13/15)

```
Process 1 (PID=<252156>): I found this trianguar number 139656
Process 1 (PID=<252156>): I found this trianguar number 140185
Process 1 (PID=<252156>): I found this trianguar number 140715
Scheduler: Runtime: 23 seconds.
Scheduler: Process 1 is running with 0 seconds left
Scheduler: Terminating Process 1 (Remaining Time: 0)
Process 1 (PID=<252156>): count of triangulars found so far is 38
Process 1: terminating....
Scheduler: Starting Process 6 (Remaining Time: 6)
Process 6 (PID=<252157>): has been started
Process 6 (PID=<252157>): will find the next trianguar number from (594687, inf)
Scheduler: Runtime: 24 seconds.
Scheduler: Process 6 is running with 5 seconds left
Process 6 (PID=<252157>): I found this trianguar number 595686
Scheduler: Runtime: 25 seconds.
Scheduler: Process 6 is running with 4 seconds left
Process 6 (PID=<252157>): I found this trianguar number 596778
Scheduler: Runtime: 26 seconds.
Scheduler: Process 6 is running with 3 seconds left
Process 6 (PID=<252157>): I found this trianguar number 597871
Scheduler: Runtime: 27 seconds.
Scheduler: Process 6 is running with 2 seconds left
Scheduler: Runtime: 28 seconds.
Scheduler: Process 6 is running with 1 seconds left
Process 6 (PID=<252157>): I found this trianguar number 598965
Scheduler: Runtime: 29 seconds.
Scheduler: Process 6 is running with 0 seconds left
Scheduler: Terminating Process 6 (Remaining Time: 0)
Process 6 (PID=<252157>): count of triangulars found so far is 4
Process 6: terminating....
Scheduler: Starting Process 2 (Remaining Time: 3)
Process 2 (PID=<252158>): has been started
Process 2 (PID=<252158>): will find the next trianguar number from (767457, inf)
```

Exercise 2 (14/15)

```
Process 2 (PID=<252158>): will find the next trianguar number from (767457, inf)
Scheduler: Runtime: 30 seconds.
Scheduler: Process 2 is running with 2 seconds left
Process 2 (PID=<252158>): I found this trianguar number 768180
Scheduler: Runtime: 31 seconds.
Scheduler: Process 2 is running with 1 seconds left
Scheduler: Runtime: 32 seconds.
Scheduler: Process 2 is running with 0 seconds left
Scheduler: Terminating Process 2 (Remaining Time: 0)
Process 2 (PID=<252158>): count of triangulars found so far is 1
Process 2: terminating....
Simulation results.....
process 0:
        at=5
        bt=2
        ct=17
        wt=10
        tat=12
        rt=10
process 1:
        at=7
        bt=6
        ct=23
        wt=10
        tat=16
        rt=10
process 2:
        at=20
        bt=3
        ct=32
        wt=9
        tat=12
        rt=9
```

Exercise 2 (15/15)

```
Simulation results.....
process 0:
        at=5
        bt=2
        ct=17
        wt=10
        tat=12
        rt=10
process 1:
        at=7
        bt=6
        ct=23
        wt=10
        tat=16
        rt=10
process 2:
        at=20
        bt=3
        ct=32
        wt=9
        tat=12
        rt=9
process 3:
        at=3
        bt=8
        ct=14
        wt=3
        tat=11
        rt=3
process 4:
        at=2
        bt=4
        ct=6
        wt=0
        tat=4
        rt=0
process 5:
        at=3
        bt=1
        ct=15
        wt=11
        tat=12
        rt=11
process 6:
        at=10
        bt=6
        ct=29
        wt=13
        tat=19
        rt=13
data size = 7
Average results for this run:
        avg_wt=8.000000
        avg_tat=12.285714
```

Exercise 3

- Modify the program **scheduler.c** and write the solution of this exercise in **scheduler_sjf.c**, in which you should implement shortest job first algorithm (non-preemptive version).
- The implementation should follow the same requirements as the previous exercise. Use the same code snippets with some modification.
- Compare the output with the outputs of the previous exercise and save the explanation in **ex3.txt**
- Submit **ex3.txt**, **scheduler_sjf.c** with a supplement script **ex3.sh** to run the program.
- **Note:** You do not need to change **worker.c**. To implement SJF here, you need to modify only the functions **find_next_process** and **schedule_handler**.

Exercise 4

- Modify the program **scheduler.c** and write the solution of this exercise in **scheduler_rr.c**, in which you should implement round-robin algorithm.
- The implementation should follow the same requirements as the previous exercise. Use the same code snippets with some modification.
- Compare the output with the outputs of the previous two exercises and save the explanation in **ex4.txt**
- Submit **ex4.txt**, **scheduler_rr.c**, and script **ex4.sh** to run the program. **Note:** for this algorithm, the quantum should be specified by the user from stdin in **scheduler_rr.c**.
- **Note:** You do not need to change **worker.c**. To implement Round robin algorithm here, you need to modify the functions **find_next_process** and **schedule_handler**. You may also need to add another field to **ProcessData** struct for storing the remaining slice of quantum.

Exercise 5 (Optional)

Design and implement a multi-core aware process scheduling algorithm that optimizes CPU utilization and minimizes response time in a multi-core system.

Exercise 6 (Optional)

- Write your own implementation of any additional scheduling algorithm you like.
- It should accept a number of processes and CPU bursts of every process.
- It should represent results as a timeline
- Example:

$$P1 ==== P2 == P3 ==== Pk === Pn$$

where '=' represents a time quantum

Default action of standard signals

Signal name	Default action
SIGINT	Term
SIGKILL	Term
SIGABRT	Core
SIGALRM	Term
SIGQUIT	Core
SIGTERM	Term
SIGSEGV	Core
SIGBUS	Core
SIGSTOP	Stop
SIGTSTP	Stop
SIGCONT	Cont
SIGUSR1	Term
SIGUSR2	Term

References

- Preemptive and Non-Preemptive Scheduling
- FCFS CPU Scheduling
- Shortest Job First (or SJF) CPU Scheduling
- Round Robin Scheduling
- Signals in C

End of lab 6